GXD verses "Direct" XML

Peter Vanderbilt*

March 20, 2000

There is a fair amount of interest in using XML (the eXtensible Markup Language [3]) for representing data. This note discusses issues regarding the standard use of XML, what we call *direct XML*, and discusses how the GXD dialect of XML addresses them. A very quick introduction to GXD is given in section 1 below; for more information, see the white paper "Uniform Data Access Using GXD" [10]. For more on XML, see www.w3.org/XML/#dev.

As we'll see, GXD puts a layer over XML which, unfortunately, makes the use of certain standard XML tools more awkward. Thus a second purpose of this note is to challenge those more knowledgeable about XML and related technologies to respond with ways to address the issues mentioned here using just standard XML (or a thinner layer). Are there standards that address some of these issues? Are later versions of XML likely to address any of these issues?

This note discusses GXD as it is envisioned, not as it is currently implemented (in the NASA-internal release of Jan 00). Not currently implemented are scripts, templates, association interfaces, interface definitions and interface versioning.

After the overview of GXD, we define our notion of "direct XML" (in section 2). Section 3 discusses issues regarding inheritance and polymorphism, section 4 discusses multiple inheritance and section 5 discusses schema evolution. We then turns to GXD's support for "indirections" (in section 6).

1 Overview of GXD

GXD consists of three things: (a) a data model with associated APIs, (b) data transport language (a dialect of XML) and (c) a library implementing the data model APIs. The data model is designed to be a common framework capable of representing a wide range of things including scientific data (or typed references to scientific data) and its metadata, associations between data items, infrastructure resources (such as users, machines and networks) and GXD metamodel items. While GXD was created in the context of scientific data management, it has wide applicability.

*MRJ Technology Solutions, NASA Ames Research Center; email: pv@nas.nasa.gov or pvanderbilt@iname.com

GXD's *data model* defines the logical notion of (GXD) *nodes.* There are four kinds of nodes: Value, List, Dictionary and Object. *Value* nodes contain data in the form of strings, integers, floats or booleans. The other nodes contain references to other nodes. A *List* node contains references to an ordered collection of other GXD nodes. A *Dictionary* (or *Dict*) node contains references to a (homogeneous) collection of other GXD nodes indexed by a key. An *Object* node contains references to a heterogeneous and extensible collection of other GXD nodes indexed by what we call *property names.* Thus nodes logically form a directed graph.

Groups of nodes, called *nodesets*, can work together to represent real-world things and nodesets can (recursively) aggregate into higher-level constructs. The total set of nodes (at any particular time) we call the *data space.*

The data model has an associated client API (in Java) which models nodes. Through the API, one can extract the content of a Value node or follow the references from List, Dict and Object nodes to other nodes. The GXD client *runtime* implements this API.

Some people call XML a "metalanguage" because it defines a set of languages, each specified by a DTD. In the same way, GXD is a metalanguage through the notion of *interfaces.* An interface defines a subset of the universe of GXD nodes and nodesets and assigns meaning to the elements of that subset. Interfaces are represented within GXD by *interface definitions* and have associated, globally-unique *interface identifiers.* All nodes contain information about their interface(s).

GXD's *data transport language* is used to communicate some region of the data space from data source to client. It uses a dialect of XML that encodes everything using a small set of tags that represent only the course structure of the data space. These tags are VALUE, LIST, DICT and OBJECT and correspond to the kinds of nodes describe previously. A VALUE element contains text (PCDATA) and is used to represent Values nodes. A LIST element contains an ordered collection of other GXD elements. A DICT element contains a collection of other GXD elements whose "id" attribute indicates the associated key. An OBJECT element contains references to a collection of other GXD elements whose "id" attribute is the property name. All elements have an "interfaceId" attribute to indicate the associated node's interface.

Consider the fragment of GXD XML given in figure 1. This fragment of

```
<OBJECT interfaceId="org.myOrg.Person">
    <VALUE id="commonName">John</VALUE>
    <VALUE id="surName">Doe</VALUE>
    <VALUE id="telephoneNumber">(555) 555-1234</VALUE>
</OBJECT>
```

Figure 1: Example of a Person instance in GXD

GXD represents an Object node of interface "org.myOrg.Person" which contains references to three Value nodes (which happen to be encoded in line). The element has the "OBJECT" tag indicating the kind of node. The "interfaceId" attribute indicates that node implements the "org.myOrg.Person" interface. Its contents are three other elements, each directly encoding one of the referenced nodes and whose "id" attribute denotes the element's property name. Each of these elements encodes a Value node.

GXD employs a few additional tags, LINK and SCRIPT, that indicate that the data is encoded by rule rather than being directly present. GXD also has an attribute for specifying a "template", an entity containing defaults. And finally, GXD has a few standard attributes that support "introspection", the ability to find out about the type and implementation of a data item.

The GXD *client runtime* interprets GXD documents and yields (Java) objects representing the data. A client application (one using the client APIs) sees the data space as a globally interconnected set of nodes – the runtime provides transparency over location and data source implementation by dealing with the indirections (LINK and SCRIPT) and by handling template references. More will be said about indirections in section 6. For more on GXD in general, see its white paper [10].

2 Direct XML

The data of figure 1 could also be represented in XML along the lines of figure 2. In this case the "Person" tag indicates the type of the entity being represented

```
<Person>
    <commonName>John</commonName>
    <surName>Doe</surName>
    <telephoneNumber>(555) 555-1234</telephoneNumber>
</Person>
```

Figure 2: Example of a Person instance in "direct" XML

and the other tags ("commonName", "surName" and "telephoneNumber") indicate properties of that entity.

This style of XML we call *direct XML*. In this style, the tag of an element indicates the element's type and possibly the role of the element within its parent. Note that, in contrast, GXD carries type indication in one attribute ("interfaceId") and role information in another ("id"); a GXD tag does imply some type information, but only at a very high level.

3 Inheritance and Polymorphism

So now consider what happens if one wants to define an extension of Person, say OrganizationalPerson, which has additional properties like "organizationalUnit" and “location”. In GXD, an instance of OrganizationalPerson could be given as in figure 3. Note that the “interfaceId” attribute contains two interface

```
<OBJECT interfaceId="org.myOrg.Person,org.myOrg.OrganizationalPerson">
    <VALUE id="commonName">John</VALUE>
    <VALUE id="surName">Doe</VALUE>
    <VALUE id="telephoneNumber">(555) 555-1234</VALUE>
    <VALUE id="organizationalUnit">My Organization</VALUE>
    <VALUE id="location">Some Building/Room</VALUE>
</OBJECT>
```

Figure 3: Example of an OrganizationalPerson instance in GXD

identifiers, one for each interface implemented. And note that the additional properties are simply included with the OBJECT.

A client that understands data of the “Person” interface, but not of “OrganizationalPerson”, can interpret this data by simply ignoring the additional properties. However a client that understands the “OrganizationalPerson” interface can interpret this same data and take advantage of the full set of properties. This feature, the ability to have one data item represent data at several levels of abstraction simultaneously, we call *polymorphism* and is one of the key design goals of GXD. In particular, an interface definition (for an Object) imposes constraints on some set of properties while saying nothing about properties outside that set.

To do the same thing in direct XML, one would first need to decide on the tag for the element. If the tag is “OrganizationalPerson”, then a client understanding only the “Person” element type will not recognize the element. But if the tag is “Person”, then including the additional elements violates the corresponding element type. Even if the additional elements are allowed (by giving up on “valid” XML), a client understanding OrganizationalPerson does not have a concrete way to know that this data item does in fact implement that extended element type.

Essentially an XML document is of a single type as specified by its DTD and each element within the document is of a single type as specified by its tag. While XML namespaces [2] allow an element type to be specified once and used in many DTDs, it is still the case that each element has a single type. In contrast, type indication in GXD is via a multi-valued attribute, so a node can be of several types.

Some proposed XML schema languages allow for inheritance. In particular, SOX (Schema for Object-Oriented XML) [6] allows one to define, for instance, an element type for OrganizationalPerson by inheriting from the element type for Person. But this results in two different element types and any given element must be of one of those two types, not both. This doesn't provide the polymorphism that GXD provides.

One way to represent the data of figure 3 using direct XML is to have two documents, one using the “Person” element type and one using “OrganizationalPerson", and to arrange to serve them to clients based on the client's capabilities. One problem with this scheme is that a single real-world entity, like some actual person, is represented by more than one document and, so, is identified by more than one URL. This makes it hard, for instance, to put a single URL in the person's directory entry or in their email signature block. In general, each real-world entity would have to have multiple URLs and there would need to be enough metainformation to know which one to use when.

4 Multiple Inheritance

A similar problem arises with multiple inheritance. For example, GXD might have a standard interface, say gxd.core.DataInfo, that contains properties for things like update time, owner and access rights. An instance might be as in figure 4. Note that the last two elements of the OBJECT have "id" attributes

```
<OBJECT interfaceId="org.myOrg.Person,org.myOrg.OrganizationalPerson,
                     gxd.core.DataInfo">
    <VALUE id="commonName">John</VALUE>
    <VALUE id="surName">Doe</VALUE>
    <VALUE id="telephoneNumber">(555) 555-1234</VALUE>
    <VALUE id="organizationalUnit">My Organization</VALUE>
    <VALUE id="location">Some Building/Room</VALUE>
    <VALUE id="gxd.core.DataInfo:owner">Some-URL</VALUE>
    <VALUE id="gxd.core.DataInfo:updateTime">
        01 Jan 2000 12:00:00 PST
    </VALUE>
</OBJECT>
```

Figure 4: Example of multiple inheritance in GXD

that use a compound name consisting of an interface id and property name separated by a colon. While not required for this example, these compound names illustrate that a property name can be kept unambiguous by having an appropriate interface id prefix. In general, compound names allow an OBJECT node to implement an arbitrary set of interfaces.

Since XML requires each element to have a unique type, it seems that to do the same thing in direct XML, one would need element types for "Person", "PersonWithDataInfo", "OrganizationalPerson" and "OrganizationalPersonWithDataInfo". As the number of potential mix-in interfaces increases, the number of element types needed to represent all combinations would increase exponentially.

5 Schema Evolution

An important consideration for any data model is that of schema evolution: what happens when one needs to change the type or structure of data?

GXD facilitates schema evolution in three ways. First, because of the general rule that a client ignores properties it doesn't understand, an interface designer can always add new optional properties to an interface (subject to rules not described here and without changing its revision level as described shortly). An optional property is one that might or might not be present in any given instance. Note that "old" data (data published without knowledge of the new properties) is still valid and usable by "new" clients (those understanding the extended interface). Similarly, new data is usable by old clients (because they ignore the new stuff).

As far as I can tell, XML assumes that DTDs and element types are constant over location and time. It seems that a change to a DTD requires simultaneous changes to all data sources and clients using that DTD. This is a hard thing to accomplish in widely distributed, autonomous environments.

The second way GXD supports evolution is via what we call *interface versioning*: associated with every interface identifier is actually a family of interfaces indexed by numbers starting at 0. When one first defines an interface, revision 0 is defined. Later one can create new revisions (numbered consecutively) and add new properties or obsolete old ones (again subject to rules not described here). Each revision is logically a separate interface and it is possible for a node to implement a range of revisions. Thus a data source can choose to implement a revised interface without dropping support for the old interface (until some appropriate interval has elapsed). Each element announces the range of interface revisions it supports and so a client can determine the maximum revision in common between it and the data source and operate on the data at that level. So again new clients can use old data and old clients can use new data (at least for a while).

I'm not aware of any facility like this directly supported by XML.

The third way GXD supports evolution is via multiple inheritance (as described earlier). Let's say a data publisher has published his data using some given interface. He then discovers that other data publishers are publishing similar data but using a different interface. He can than add support for that interface to his data by using multiple inheritance. (And if his customers stop using his data via the old interface, support for it can be dropped.)

As discussed earlier, XML doesn't support multiple inheritance and, so, it doesn't support this form of evolution.

One of the consequences of GXD's support for evolution is that it supports organic, evolutionary growth of data sources and their standards. This is in contrast to what we call "standardization by committee" where a bunch of people get together and try to come up with a single, immutable standard that they can all use. This is the model to use when one can only implement one standard and when changing the standard breaks current implementations. Because there's only one standard and it's pretty much immutable once defined, everyone tries to get everything in it that they might possibly want and everyone has to agree on every little detail. And all too often this is done without having implemented and used real software based on the standard.

In contrast, GXD supports *evolutionary standardization*. GXD allows multiple interfaces (GXD's kind of standard) to be in use simultaneously. GXD interfaces will undergo "natural selection" in the sense that good ones will be widely used while poor ones will be dropped over time. Furthermore, interfaces can evolve via interface versioning and inheritance, even while in use. Since an interface can be extended, it needs not have every conceivable property – it can be defined with those properties known to be needed at first and others added later. And if a property is defined incorrectly, an improved one can be defined in a later revision. We believe that this sort of standardization process will lead to much better interfaces over time, as compared to the standardization by committee process.

6 Indirections

Section 1 mentioned that the GXD data language includes a handful of additional tags that are used to encode data by rule. One such tag is LINK. A LINK element has a "ref" attribute that contains a URL (possibly relative to the current document). The GXD runtime, upon encountering a LINK element, obtains the document named by the URL and returns the contained GXD node in place of the LINK element. Since LINK can contain arbitrary URLs, accessing a URL may involve program execution at the data source. Also, the URL can contain a fragment identifier that is used to refer within the named GXD entity.

The LINK mechanism has many uses. For one, it allows one to break a big result into a number of smaller pieces which are obtained on demand transparently by the runtime. Thus a data source could potentially serve up an entire database as one logical entity by yielding a top-level GXD node with LINKs for further data.

LINKs also allow for a kind of referral where a result can indicate that all or part of the result is found at some other server. This allows data to be distributed across several servers and it allows data to migrate transparently to the client.

LINKs also facilitate the sharing of data because many documents can have LINKs to the same node. This can save space because the same data need not be replicated in every document. But perhaps more importantly, it allows for a single point of presence for each instance – one can extend the data by adding properties and interfaces in just one place and all documents linking to that node get the changes.

When encoding directly in XML, one can reference another document using URLs [1], XML pointers (XPointer [7] which is built on XPath [5]), XML links (XLink [8]) or other mechanisms. But any of these indirections must be handled by the client. By having the GXD runtime handle indirections instead, the client is simplified. Another advantage is that the placement of LINKs can change over time without requiring changes in the corresponding clients.

GXD also has application-visible links via the "Association" interfaces. These standard interfaces, which are similar in functionality to that of XLink [8], represent associations, references and relationships. They allow applications to employ hyperlinks or other rendering mechanisms.

Another kind of indirection planned for GXD is the SCRIPT element, which allows (JavaScript) program text to be included in a GXD file. The client runtime replaces each SCRIPT element with the GXD node subtree generated by running that script.

It is also planned for GXD to have the notion of templates. A *template* is a GXD nodeset that contains default information for a set of nodes, including the templates for its children. A node that refers to a template has that template's default properties logically interpolated into it, except for properties already present in the node. This allows a data source to factor out information common to a collection of nodes, thus saving space and transmissions costs. Perhaps more importantly, it should alleviate the pressure to skimp on per-node information thus allowing each major node data item to be complete and self contained.

We believe that the combination of links, scripts and templates will allow encoding techniques far beyond what one would get using direct XML encoding. For instance, we expect that some classes of multiple inheritance can be implemented entirely using templates with links and scripts – the actual, pre-template data need not change at all. In general, the features of GXD allow for a wide range of data source implementations and allow data sources to evolve without affecting existing clients.

7 GXD and XML Interoperability

Although GXD and direct XML have the differences mentioned in the last few sections, we expect them to be interoperable. We have prototyped XSLT [4] translators from XML to GXD. We suspect it's possible to employ GXD SCRIPT elements with (DTD-specific) programs that convert XML to GXD nodes on the fly, thus allowing XML documents to be "injected" into the GXD data space.

Going from GXD to XML is slightly harder because standard tools like XSLT may have difficulty with GXD's indirections (links, scripts and templates) – they don't have GXD's runtime built in. Also, because of the difference in models, one GXD entity may map to several XML documents (one per interface implemented). However, technology like JSP (Java Server Pages [9]) should be able to operate over the the GXD runtime, thus leveraging the full power of GXD.

8 Summary

GXD is defined over XML but uses XML in a non-standard way. In "direct" XML, tags are used to indicate type or identity. In GXD, only a handful of GXD-defined tags are used and they define the rough structure of the data. Indications of type and identity are carried in attributes. GXD allows an element to have multiple types, thus supporting polymorphism, multiple inheritance and interface versioning. These features in turn facilitate evolutionary standardization, which should help a community to converge on the best standards. In contrast, direct XML restricts an element to have only a single type and, so, makes it difficult to support polymorphism, multiple inheritance, interface versioning and evolutionary standardization.

The GXD data format also allows for links, scripts and templates. The GXD runtime interprets these things and yields logical nodes, instances of a standard API, that are independent of location, distribution and data source implementation. In contrast, direct XML has no such facilities and, so, any similar functionality is relegated to the application itself.

References

[1] T. Berners-Lee, L. Masinter, and M. McCahill. *Uniform Resource Locators (URL), RFC 1738.* Network Working Group of the IETF, Dec 1994. http://info.internet.isi.edu/in-notes/rfc/files/rfc1736.txt.

[2] T. Bray, D. Hollander, and A. Layman. *Namespaces in XML.* World Wide Web Consortium (W3C), rec-xml-names-19990114 edition, Jan 1999. http://www.w3.org/TR/1999/REC-xml-names-19990114/.

[3] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. *Extensible Markup Language (XML) 1.0.* World Wide Web Consortium (W3C), rec-xml-19980210 edition, Feb 1998. http://www.w3.org/TR/REC-xml.

[4] J. Clark. *XSL Transformations (XSLT), Version 1.0.* World Wide Web Consortium (W3C), rec-xslt-19991116 edition, Nov 1999. http://www.w3.org/TR/xslt.

[5] J. Clark and S. DeRose. *XML Path Language (XPath).* World Wide Web Consortium (W3C), rec-xpath-19991116 edition, Nov 1999. http://www.w3.org/TR/xpath.

[6] A. Davidson, M. Fuchs, M. Hedin, M. Jain, J. Koistinen, C. Lloyd, M. Maloney, and K. Schwarzhof. *Schema for Object-Oriented XML 2.0.* World Wide Web Consortium (W3C), note-sox-19990730 edition, July 1999. http://www.w3.org/TR/NOTE-SOX/.

[7] S. DeRose, R. Daniel, and E. Maler. *XML Pointer Language (XPointer).* World Wide Web Consortium (W3C), wd-xptr-19991206 edition, Dec 1999. http://www.w3.org/TR/xptr.

[8] S. DeRose, E. Maler, D. Orchard, and B. Trafford. *XML Linking Language (XLink)*. World Wide Web Consortium (W3C), wd-xlink-20000221 edition, Feb 2000. http://www.w3.org/TR/xlink/.

[9] Sun Microsystems, Inc. *JavaServer Pages – Dynamically Generated Web Content*. http://www.java.sun.com/products/jsp/.

[10] P. Vanderbilt. *Uniform Data Access Using GXD*. Numerical Aerospace Simulation Systems (NAS) Division of NASA, Sept 1999. http://www.nas.nasa.gov/~pv/gxd/.